COMP 3000: Operating Systems

Carleton University
Fall 2018 Test 1 Solutions

1. [1] Is it possible for two or more filenames to refer to the same inode? Explain briefly.
   A: Yes it is possible—this is exactly what hard links are.

2. [2] In 3000pc, every entry in the queue has a lock associated with it. If these per-entry locks were replaced by a single lock for the entire queue, could the program still execute correctly (enforcing proper mutual exclusion)? Would the performance be the same?
   A: You could use just one lock for the entire queue, and it would execute correctly. However, performance would suffer because the single lock would prevent the producer and consumer from writing to the queue concurrently.

3. [2] /dev/random and /dev/urandom:

   (a) [1] /dev/random can only produce a relatively small number of bits at a time. Why is it so slow?
   A: It is slow because it derives bits external sources of entropy (physical/external processes), rather than by running an algorithm.

   (b) [1] In contrast, why can /dev/urandom (and random() from the C library) produce an essentially unbounded number of bits very quickly?
   A: /dev/urandom can be much faster because its bits are produced by an algorithm (a cryptographically-secure pseudo-random number generator). It only has to be seeded using external sources of entropy.

4. [4] Signals:

   (a) [1] How can a process send a signal?
   A: A process sends a signal by executing a kill system call.

   (b) [1] How can a process change what happens when a signal is received?
   A: A process can make a sigaction system call to register a new signal handler.

   (c) [1] Can a process change what happens when any signal is received?
   A: No, some signals such as KILL and STOP do not have customizable signal handlers—their effects are defined by the kernel.

   (d) [1] When a process receives a signal, what happens to the function that the process was already running?
   A: The function is paused (its current state is pushed onto the function call stack) and it resumes from where it left off once the signal handler finishes.

5. [4] Recall that 3000shell.c sets the path variable using the following code:

```
char *default_path = "/usr/bin:/bin";
path = find_env("PATH", default_path, envp);
```

   (a) [1] envp contains the process's environment variables. Who sets the initial values of these variables? How?
   A: The execve system call that loaded the current program binary defines a process's environment variables. Thus, whoever made the execve call defined the environment variables. Thus, if bash runs ls, bash defines the environment variables for ls (because bash does execve("/bin/ls",...).

   (b) [1] Could 3000shell change the value of its PATH environment variable? Explain briefly.
   A: Yes, because a process can change its own environment variables because they are just values in the process's address space.

   (c) [1] What is the PATH environment variable used for normally?
   A: PATH is used to define the list of directories in which to search for programs to run that are specified at a shell prompt. PATH is what allows the shell to find the ls command in /bin.

   (d) [1] Why does 3000shell need the value of PATH?
   A: 3000shell needs PATH because it is a simple UNIX shell, and so it needs to know where to look for programs that it is asked to execute.

6. [5] Consider the following code from 3000shell.c:

```
if (background) {
    fprintf(stderr, "Process␣\%d␣running␣in␣the␣background.\n", pid);
} else {
    pid = wait(ret_status);
}
```

   (a) [1] What does 3000shell.c call before this in order to create a child process?
   A: The fork system call

   (b) [1] What does the call to wait() above do?
   A: It waits for the child process (created by fork) to exit. It returns the PID of the child process that exited. If ret_status is non-NULL, it stores the child process's exit value in int variable pointed to by ret_status. (In the code, ret_status is NULL so no return status is stored.)

   (c) [1] When does 3000shell also call wait()?
   A: It also calls wait in the SIGCHLD signal handler. The kernel sends 3000shell a SIGCHLD signal when a child process terminates. The signal handler code is necessary to take care of background processes (i.e., prevent them from becoming zombies).

   (d) [2] Does the process that executes the above code also run the program the user has typed in at the command prompt? Explain briefly.
   A: It does not. The above code runs in the main 3000shell process. Commands are execve'd in child processes. Only the child processes can call execve because otherwise 3000shell would effectively terminate by running execve.

7. [4] You are trying to port 3000pc to a system that does not provide any implementation of sem_wait() or sem_post(). You search online and you find the following implementations of sem_wait() and sem_post():

```
void sem_wait(int *sem)
{
    while (!dec_nonzero(sem))
        /* wait */
    }
}

void sem_post(int *sem)
{
    *sem++;
}
```

   (a) [1] Does the above code make any system calls?
   A: The above code makes no system calls, as it only has a while loop, an integer increment, and a function call. It is possible that dec_nonzero() makes a system call however.

   (b) [1] If the semaphore is locked (is less than zero), what does sem_wait() do, precisely?
   A: sem_wait() keeps calling dec_nonzero() until it returns a nonzero (true) value. Note that such a loop is also known as a spinlock.

   (c) [2] The code has a comment saying that you must supply an architecture-specific assembly language implementation of dec_nonzero() in order for this code to work properly. Why can't you just write dec_nonzero() in C with something like this?

```
int dec_nonzero(int *n)
{
  if (*n > 0) {
    *n--;
    return 1;
  } else {
    return 0;
  }
}
```

A: The above code is not correct because the check (the comparison in the if statement) and the decrement of *n do not occur atomically (at the same time, as a single operation). They will produce two separate instructions, making it possible for another core to change the value of *n after it has been checked but before it has been modified. If that were to happen we could have two cores take the lock at the same time—the very thing we were trying to avoid. Also, the call to dec_nonzero in sem_wait() does not pass a pointer to an integer (an ampersand is missing), so things are just broken unless dec_nonzero() is implemented as a macro.

8. [4] Standard input is the default input source for C programs which using functions such as gets() and scanf(). When answering the following questions, make sure to explain your answers.

   (a) [1] When a program reads from standard input, is it reading from a file descriptor? If so, which one?
   A: It is reading from file descriptor 0.

   (b) [1] Does a program need to open standard input before reading from it?
   A: No, standard input, standard output, and standard error (file descriptors 0, 1, and 2) can be assumed to have been opened already.

   (c) [1] Can a program change where standard input gets its data from?
   A: Yes, it simply has to close standard input, open another source for reading, and use dup2 to copy the open file descriptor to 0.

   (d) [1] At the system call level, is reading from standard input the same as reading from a regular open file?
   A: Yes, reading from standard input is the same as reading from any file—you make read system calls. The only difference is you don't have to open standard input, as it has already been opened. Note you could also use mmap, but that only works if you are reading from a non-special file. But you can open a special file on any file descriptor, so this isn't an issue with standard input or output specifically, except that they are often bound to character devices representing terminals.

9. [4] mmap() is declared as follows:

```
void *mmap(void *addr, size_t length, int prot, int flags,
            int fd, off_t offset);
```

   As we've used it in class, normally addr is set to NULL and offset is 0.

   (a) [1] At a high level, what does mmap() do?
   A: mmap associates the contents of a file with a range of memory addresses. If no file is specified, the memory is just allocated.

   (b) [1] What's the difference between setting flags to be MAP_SHARED versus MAP_PRIVATE in the context of child processes?
   A: With MAP_SHARED, all child processes will share the same memory, changes made by one will be visible to all. Also, if a file descriptor was specified, its contents will be changed when the corresponding memory is changed. With MAP_PRIVATE, all changes to memory are private. Every process has its own unique copy (with its contents being copied on fork like all other regular memory). Further, the opened file is not changed to reflect the changes to the mapped copy.

   (c) [2] If we open a file and mmap it, setting `prot=PROT_READ|PROT_WRITE` and `flags=MAP_PRIVATE`, can we modify the memory that is returned by mmap? If we modify this memory, will it modify the file as well? Explain briefly.
   A: If we set PROT_READ and PROT_WRITE, we can indeed modify the returned memory. Any changes, however, will be private if MAP_PRIVATE is specified, so the opened file will not be changed; the initial contents of memory will be the same as the file, but subsequent changes will be private to the process.